# *CISC322/326 ASSIGNMENT 3: REPORT*

GNUstep: Enhancement Proposal

*April 4, 2025*

Group 18
Chi Ma (Presenter)
Nick He
Zhiming Jin
Tiantian Sang (Presenter)
Dunyi Xie
Lixing Yang (Leader)

## Table of Contents

# 1 Abstract

This report presents the design, integration, and evaluation of an AI Code Assistant module within the GNUstep development environment, specifically targeting the Gorm IDE and ProjectCenter. The primary goal is to enhance developer productivity by enabling intelligent code suggestions and on-demand documentation based on real-time project context. The report begins by outlining the current architectural limitations of GNUstep, such as the lack of prompt-handling mechanisms, metadata extraction, and modular data exchange protocols. To address these issues, we propose and implement a loosely coupled, asynchronous AI subsystem that communicates via defined APIs without disrupting core IDE functionalities. Two sequence diagrams illustrate key use cases: context-aware code generation and tooltip-based class explanation. A SAAM analysis compares two integration approaches, ultimately recommending a modular design for maintainability, reusability, and system stability. Testing plans, impacted files, and lessons learned further validate the implementation, demonstrating that thoughtful modular architecture and robust testing are essential for successfully introducing AI features into mature software systems.

# 2 Introduction and Overview

To improve the productivity and development experience within the GNUstep environment, we propose the integration of a new module: the AI Code Assistant. This assistant leverages artificial intelligence to provide context-aware coding support, filling a critical gap in the current workflow of tools like Gorm and ProjectCenter. By delivering intelligent code suggestions, real-time documentation, and prompt-driven support, the AI Code Assistant addresses several limitations inherent in GNUstep's manual, static development processes.

**Context-Aware Code Suggestions**
The AI Code Assistant enhances Gorm IDE by offering dynamic Objective-C code snippets based on the current UI layout and development context. When a developer interacts with UI elements, the assistant analyzes their properties and placement to generate relevant, editable code, reducing the need for manual boilerplate and accelerating the prototyping phase.

**On-Demand Documentation and Tooltip Explanations**
Developers can hover over unfamiliar classes or components and receive immediate explanations and usage examples in the form of tooltips. This minimizes context-switching and streamlines the learning curve, especially for newcomers to Objective-C or GNUstep.

**Modular and Maintainable Architecture**
Rather than embedding the assistant directly into core IDE components, we designed it as a modular subsystem with well-defined APIs. This approach preserves GNUstep's clean separation of concerns, enabling independent development, testing, and scaling of the assistant without disrupting existing workflows.

**Developer-Centric Enhancements**
The assistant introduces support for preference settings, undo/redo integration, and UI feedback mechanisms—ensuring it remains helpful without being intrusive. Additionally, it improves

debugging and testing through detailed logging and structured interaction records, empowering developers to fine-tune both their code and the assistant itself.

In conclusion, the integration of the AI Code Assistant is a well-motivated architectural enhancement to GNUstep. It bridges the gap between UI design and code implementation, enriches developer tooling with intelligent automation, and lays the foundation for a more efficient, scalable, and modern development environment.

# 3 Recap of Conceptual Architecture

## 3.1 High-level Conceptual Architecture with AI Consistent Module

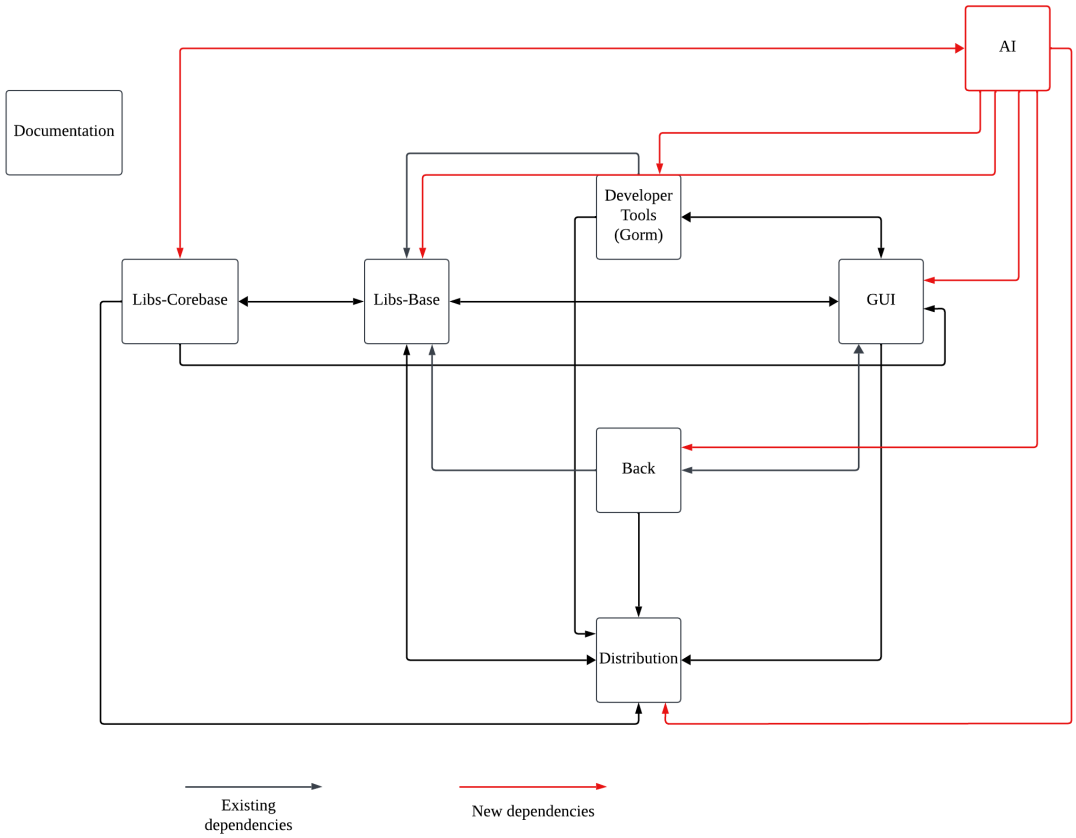



**Figure 1: High-level conceptual architecture**

AI Code Assistant → Developer Tools
The AI assistant integrates into Gorm or other GNUstep IDEs, leveraging the Developer Tools' internal APIs to gain real-time knowledge of project structures (e.g., class files, resources, build settings). This allows the assistant to generate code snippets or refactor suggestions based on the developer's current context, displaying them directly in the IDE without needing modifications in the Tools themselves.

AI Code Assistant → Base
For file I/O, string manipulation, and concurrency support, the assistant relies on Foundation (Base). It reads UI layouts from disk, manages JSON or XML requests to an external LLM if

needed, and uses concurrency primitives to run background inference. Base does not depend on the assistant, so communication flows only in one direction.

AI Code Assistant → GUI
The assistant uses GUI classes to render its suggestions and overlays. For example, when a developer hovers over a UI component in Gorm, the assistant may present a tooltip with recommended Objective-C code. This one-way dependency enables visual integration in the design environment, while the GUI layer itself remains independent of AI features.

AI Code Assistant → Back
Certain functionality—like drawing dynamic overlays, highlighting code, or intercepting low-level UI events—may require calls to the Back layer. Since Back handles the platform-specific rendering, the assistant instructs it on how to display suggestion pop-ups or code highlights. The Back layer has no need for AI features, so the dependency does not extend in reverse.

AI Code Assistant ↔ CoreBase
CoreBase provides specialized services such as cryptographic operations (e.g., encrypting local data for privacy) or advanced data handling libraries, which the assistant can leverage. In turn, CoreBase may use AI-driven features—for instance, code generation routines or sophisticated pattern matching. This two-way relationship expands both components' capabilities.

AI Code Assistant → Distribution
When finalizing builds, the assistant's resources—like local model files or hooks to an external LLM—must be packaged with the overall GNUstep application. Distribution scripts place the AI assistant's modules within the correct bundle or installer, but they do not otherwise rely on the assistant for their own functions, thus making it a one-direction dependency.

**3.2 Low-Level Conceptual Architecture with the AI Code Assistant module**

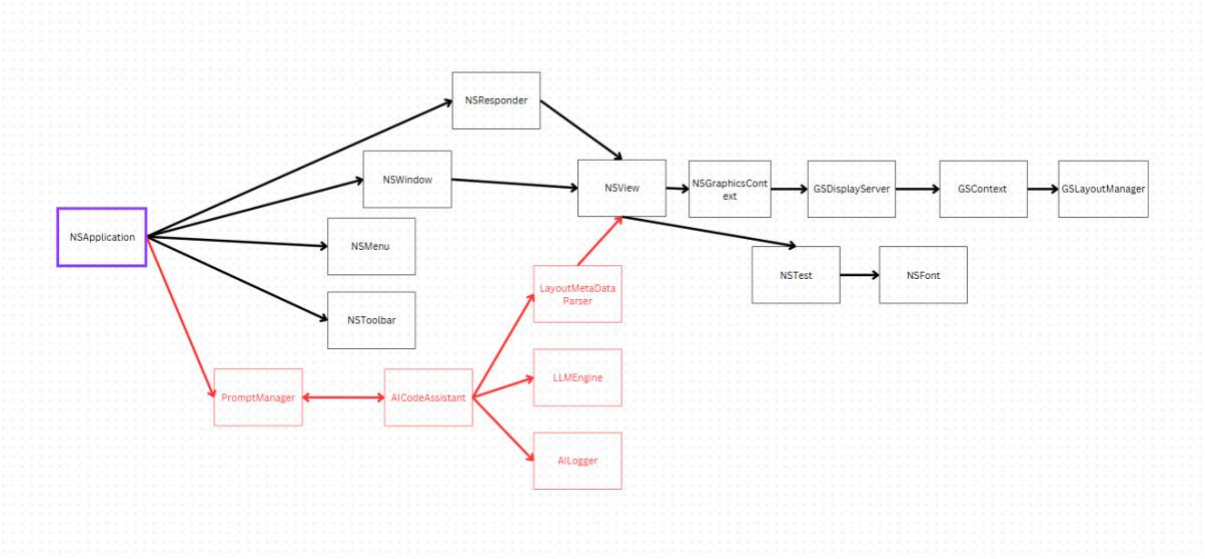



**Figure 2: Low Level conceptual architecture**

The AI Code Assistant module is designed to work alongside the existing GNUstep libs-gui structure without disrupting core GUI functionalities. It adds a new layer of intelligent code suggestion capabilities, driven by layout context or natural language input from the developer.

PromptManager:
Main Responsibilities: Acts as the bridge between the developer and the AI module. It captures text-based prompts and UI context events and sends them to the AI assistant.
Dependencies: Connected to NSApplication (for prompt triggers and UI state), and communicates bidirectionally with AICodeAssistant.

AICodeAssistant:
Main Responsibilities:Processes the prompt and layout information, communicates with the LLM to generate Objective-C code, and manages AI-related operations.
Dependencies: Receives input from PromptManager, fetches context via LayoutMetadataParser, sends requests to LLMEngine, and logs results via AILogger.

LayoutMetadataParser:
Main Responsibilities: Extracts UI layout structure and component metadata from NSView and related components.
Dependencies: Called by AICodeAssistant to access the NSView hierarchy for context-based suggestions.

LLMEngine:
Main Responsibilities: A backend service (local or remote) that processes prompts and returns AI-generated Objective-C code snippets.
Dependencies: Only receives requests from AICodeAssistant, does not depend on GUI components.

AILogger:
Main Responsibilities: Stores input prompts, layout context, and AI responses for debugging, feedback collection, and system improvement.
Dependencies: One-way interaction from AICodeAssistant, fully decoupled from the GUI.

## 4 The Current State of the System Relative to the Enhancement

Currently, The GNUstep's workflow is manual, where developers create interfaces in Gorm by placing UI elements such as text fields, buttons, and labels. After designing the UI, developers manually implement Objective-C code logic corresponding to elements within ProjectCenter or other code editors. While focusing on static interactions between its IDE components, the system lacks automated or dynamic mechanisms that bridge UI design and code generation.

At a general architectural level, Gorm primarily interacts with UI components like NSApplication and NSView, handling their instantiation, layout, and properties within an application window. The editor components within ProjectCenter manage code editing, project structuring, and build processes. Those components are optimized for manual code management through synchronous direct method calls and file manipulations. They lack built-in features to

capture developer prompts, provide integrated AI support and context-aware code suggestion capability, or extract real-time UI layout metadata.

**Key Limitations Affecting AI Integration:**

- Absence of Metadata Extraction:
  Currently, UI components like NSView do not have abilities to extract structured metadata. For example, when developers place or modify UI elements in Gorm, no real-time contextual data (such as element type, attributes, positioning, or hierarchical relationships) is exported for further use. This limitation prevents automatic or AI-assisted analysis of UI layouts.

- Lack of Prompt Management Components:
  IDE tools like Gorm and ProjectCenter currently provide no structured mechanism to capture or interpret natural language developer prompts or context-specific intentions.

- Synchronous Communication:
  IDE can freeze or become unresponsive during long operations due to synchronous communications with supporting libraries. Developers may encounter latency or blocking behavior when handling large UI layouts or extensive project files, and these issues get considerably worse when AI integrations are used.

- Missing Data Exchange Standards:
  GNUstep currently lacks standardized formats like JSON or XML for inter-module communication. Each IDE component exchanges data through custom or ad-hoc structures, making it challenging to add external AI-driven module due to manual effort and custom adaptions.

The current limitations of the GNUstep system negatively affect maintainability, developer productivity and creativity. Rapid prototyping is slowed down, coding errors are increased, and the development lifecycle is prolonged when manual coding is used. Lacking automatic code generation also restricts rapid experimentation and prevents developers from easily testing multiple UI layouts or quickly applying application logic. System responsiveness is also a concern—integrating features like AI-assisted coding could significantly degrade IDE performance and causes reduced usability. Finally, Integration of new features is complicated due to insufficient data exchange protocols. This makes the system harder to evolve, maintain, and scale.

The enhanced improvement, on the other hand, uses an architecture that is loosely coupled, asynchronous, and modular, decoupling components to allow for independent development, testing, and deployment. Integration is streamlined by standardized communication protocols and clearly defined interfaces. Isolated components improve system reliability by localizing failures, automated code generation. Additionally, the modular design encourages maintainability and scalability, enabling resource allocation and updates that are flexible without causing system-wide disruptions. When combined, these enhancements produce a robust, flexible framework that puts efficiency, stability, and future expansion first.

# 5 SAAM Architecture Analysis

**Step 1: identifying main stakeholders:**

By examining both the high-level and low-level conceptual architecture of the enhanced GNUstep system, we identify the major stakeholders involved in the AI Code Assistant integration as follows:

- **GNUstep Developers:** Core maintainers of GNUstep frameworks, including libs-gui, libs-back, and Gorm. They are responsible for incorporating the new AI module into the development environment in a stable, maintainable way.

- **GNUstep Users (Application Developers):** These are developers using GNUstep IDEs (like Gorm or ProjectCenter) to build Objective-C applications. They directly interact with the AI assistant to generate code snippets and improve development productivity.

- **Project Managers:** Oversee delivery timelines, coordinate communication between core GNUstep developers and AI integrators, and ensure that the integration aligns with project goals.

- **AI Integrators:** Responsible for developing and maintaining the AI assistant backend, including integration with local or external LLM APIs, ensuring flexibility and upgradability of the AI functionality.

**Step 2: Stakeholder NFRs**

Each stakeholder has specific non-functional requirements that must be considered in the architecture of the enhanced system:

- GNUstep Developers require the AI module to be loosely coupled with existing components to maintain a clean architectural structure. It should also be easy to test, well-documented, and robust to minimize the risk of introducing instability into core modules.

- GNUstep Users expect the AI assistant to be responsive, accurate, and non-intrusive. It should improve their productivity without slowing down the IDE or disrupting the interface. It should also be optional, allowing users to enable or disable it based on personal preference.

- Project Managers value predictable delivery, smooth integration, and low-risk deployment. Clear documentation, modular structure, and effective communication between teams are also important.

- AI Integrators need the module to support modularity, LLM back-end independence, and easy upgrades. This allows them to iterate or improve the assistant's capabilities without requiring changes to the GNUstep core system.

**Step 3: Evaluating Impacts of Each Approach**

1st Approach:

This approach proposes embedding the AI assistant directly within the existing Gorm or ProjectCenter IDEs. Prompt management, layout parsing, and LLM communication would be

handled internally by the IDE, integrating the assistant tightly into the GUI layer. This approach simplifies the initial development process and can reduce latency by avoiding inter-module communication.
However, this integration introduces significant architectural risks. It couples the AI logic with GUI components such as NSView, NSApplication, and editor panels. This violates the principle of separation of concerns, complicates future testing and maintenance, and increases the risk of regressions in the existing GNUstep GUI subsystems. Updating the AI backend or replacing the LLM would likely require changes across multiple parts of the IDE, making the system more fragile and harder to evolve.

2nd Approach:
This approach proposes implementing the AI assistant as a separate subsystem outside the GUI framework, communicating via a defined interface such as a PromptManager API. This modular design ensures that the assistant operates independently and can be integrated not just into Gorm, but also into other GNUstep development tools.
While this approach requires more effort up front to design robust communication mechanisms, it greatly enhances maintainability, supports independent deployment, and allows the AI system to evolve or scale without impacting core GNUstep functionality.

**Step 4: Finding the Best Path to Realize the Enhancement**
After reviewing the two implementation approaches and weighing their impact against the identified NFRs, the following table summarizes their effects:

| NFRs | Effect (Approach 1) | Effect (Approach 2) |
|---|---|---|
| Maintainability | Negative (requires modification to core GNUstep files) | Positive (only AI module needs updating) |
| Evolvability | Neutral (evolution limited by integration complexity) | Positive (modular design allows independent evolution) |
| Testability | Neutral (embedded in GUI codebase) | Positive (modules can be tested in isolation) |
| System Performance | Positive (minimal inter-module delay) | Neutral (small overhead from module communication) |
| Reusability | Negative (bound to one tool, like Gorm) | Positive (can be reused across Gorm, ProjectCenter, etc.) |
| Stability | Negative | Positive |

| | | |
|---|---|---|
| | (tight coupling increases regression risk) | (isolated logic reduces impact on core system) |

Through this comparison, we conclude that Approach 2: Modular AI Assistant is the better solution. It provides a cleaner, future-proof architecture that respects GNUstep's existing layering, supports independent module testing and deployment, and minimizes risk during integration. Although it introduces slightly more complexity in the initial design phase, the long-term benefits in scalability, maintainability, and system integrity are significantly greater.

Therefore, our team recommends implementing the AI Code Assistant as an independent subsystem, interfaced through well-defined APIs, to maximize architectural quality and stakeholder satisfaction.

# 6 Impacted Files

To add the AI Code Assistant to GNUstep (especially in Gorm and the development tools), we found that several files need to be updated or extended. These changes help send information from the UI to the assistant and show suggestions to the user.

## 6.1 Gorm

- GormController.m / GormController.h

This file handles the main logic in Gorm. We need to add new functions here to collect user input (like selected UI elements) and send it to the AI assistant. Also, when the assistant gives back code, this file can handle how to insert the code into the right place.

- GormDocument.m / GormDocument.h

GormDocument manages the current project being edited. We will update it to display the code suggestion and make sure the user can accept or reject the result. It should also update the document state (like undo/redo) after inserting the code.

- GormPreferences.m / GormPreferences.h

This file should include new settings for the AI assistant, like turning it on or off, or changing how often it gives suggestions.

## 6.2 GUI Library (libs-gui)

- NSView.m / NSView.h

NSView shows the user interface elements. We may need to add code to collect extra information from UI elements, like type or name, so the assistant can give better code suggestions.

## 6.3 Core Library

- CodeAssistantRequest.m / CodeAssistantRequest.h(new)

A new class that helps send information to the AI assistant, prepare the request, and wait for the result.

- CodeAssistantResponse.m / CodeAssistantResponse.h(new)

This file will take the response from the assistant and change it into code format. It can also help check for errors or strange outputs.

### 6.4 ProjectCenter (IDE part)

- ProjectFileEditor.m / .h

If we want to allow code suggestions while editing code directly (not only in Gorm), we can update this file to support inserting AI suggestions.

- ProjectCenterMenu.m / .h

Add new menu options like “Ask AI for Help” or “Generate Method” so users can use the assistant from the interface easily.

## 7 Architecture Analysis

In the Client–Server architectural style, the AI module, which can also power the AI Code Assistant for Objective-C, is hosted on a separate process or remote infrastructure. The GNUstep-based GUI then acts purely as the client, routing user requests to the back-end via a defined protocol (for instance, REST or RPC). When a developer requests code suggestions or runs inference, the front end packages this data and sends it to the dedicated server, where the AI logic, including the language model, processes it. This division makes the system more flexible and easier to maintain because you can independently update or scale the AI. If you need to deploy a more powerful model for the Code Assistant, you can do so without modifying the GNUstep application; switching servers or upgrading hardware is handled on the back end, while the user interface continues to operate seamlessly.

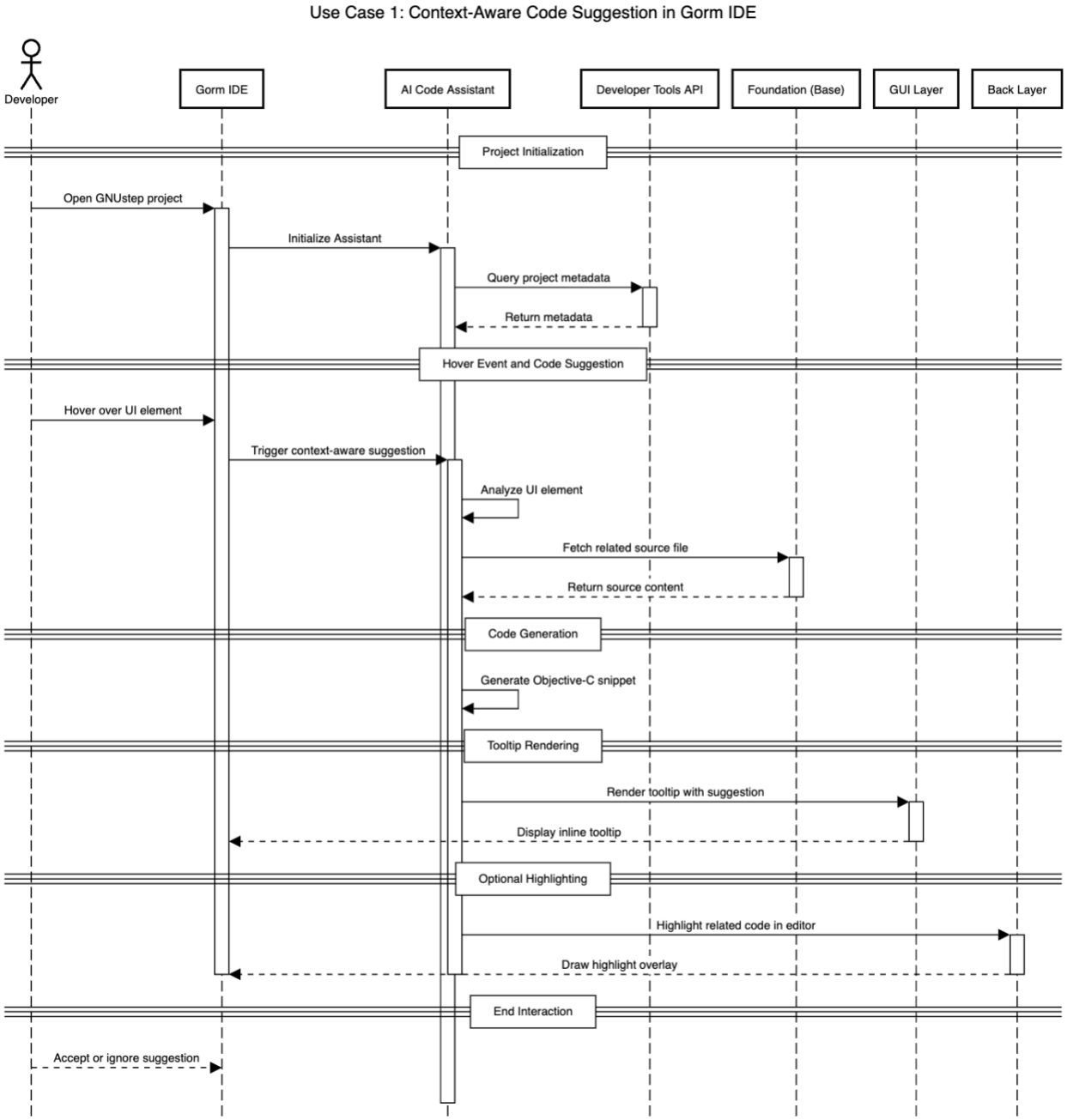
In the Layered architectural style, the entire solution remains in one application, but responsibilities are clearly separated across distinct layers. The presentation layer, which manages the GNUstep UI, captures interactions and passes them to the business logic layer, where AI tasks—such as suggesting Objective-C code snippets or performing inference—are handled. Below that, a data access layer looks after tasks like model loading, logging user interactions, or saving generated code patterns for later use. This organization helps maintain a clear boundary between user interface concerns, AI functionality, and data handling. If the AI model for the Code Assistant needs refinement, developers can focus on the business logic layer without significantly affecting how the UI is rendered or how data is stored, resulting in a cleaner, more maintainable architecture overall.

## 8 Use Case

### 8.1 Use Case 1: Context-Aware Code Suggestion in Gorm IDE

The sequence diagram of Use Case 1 illustrates how the AI Code Assistant integrates with the GNUstep development environment to provide real-time coding support. The process begins when the Developer opens a project in Gorm IDE, which then initializes the AI Assistant. The assistant queries project metadata from the Developer Tools API to gain contextual awareness of the current development environment. When the developer hovers over a UI element, the IDE triggers the assistant to generate a code suggestion. To do this, the assistant analyzes the UI element and fetches relevant source files using Foundation (Base). After retrieving the necessary content, it generates an appropriate Objective-C snippet. This suggestion is rendered as a tooltip via the GUI Layer, providing immediate visual feedback to the developer. Optionally, the Back Layer is used to highlight related code in the editor. Finally, the developer can either accept or

ignore the suggestion, completing the interaction. The diagram clearly shows the sequence of activations and the flow of control between components, reflecting a modular and responsive design architecture.



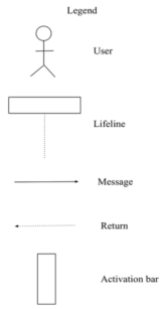



**Figure 3: UML Sequence Diagram of Use Case 1**

### 8.2 Use Case 2: Tooltip Explanation for Unknown Class

The sequence diagram demonstrates how the AI Code Assistant provides on-demand class documentation within the GNUstep Gorm IDE. The process begins when a Developer hovers over a class name in the editor—such as NSOperationQueue. This triggers the Gorm IDE to request a class explanation from the AI Assistant. In response, the assistant initiates a Context Analysis phase by querying the Developer Tools API to retrieve information about the class reference, such as its source file location and line number.

After receiving the contextual data, the assistant analyzes the class to determine whether a predefined explanation, documentation snippet, or example usage can be provided. Once the analysis is complete, the assistant proceeds to the Tooltip Display phase, instructing the GUI Layer to visually render a tooltip containing the class information directly in the editor. Optionally, to enhance clarity, the assistant may also engage the Back Layer to draw a visual highlight on the hovered class name—indicating the target of the tooltip.

The interaction concludes with the tooltip displayed and the developer free to continue coding, having received just-in-time assistance without leaving their development flow. The diagram illustrates a clean separation of responsibilities, emphasizing a non-intrusive, supportive AI assistant that leverages the IDE's internal APIs and rendering layers without altering their core behavior.

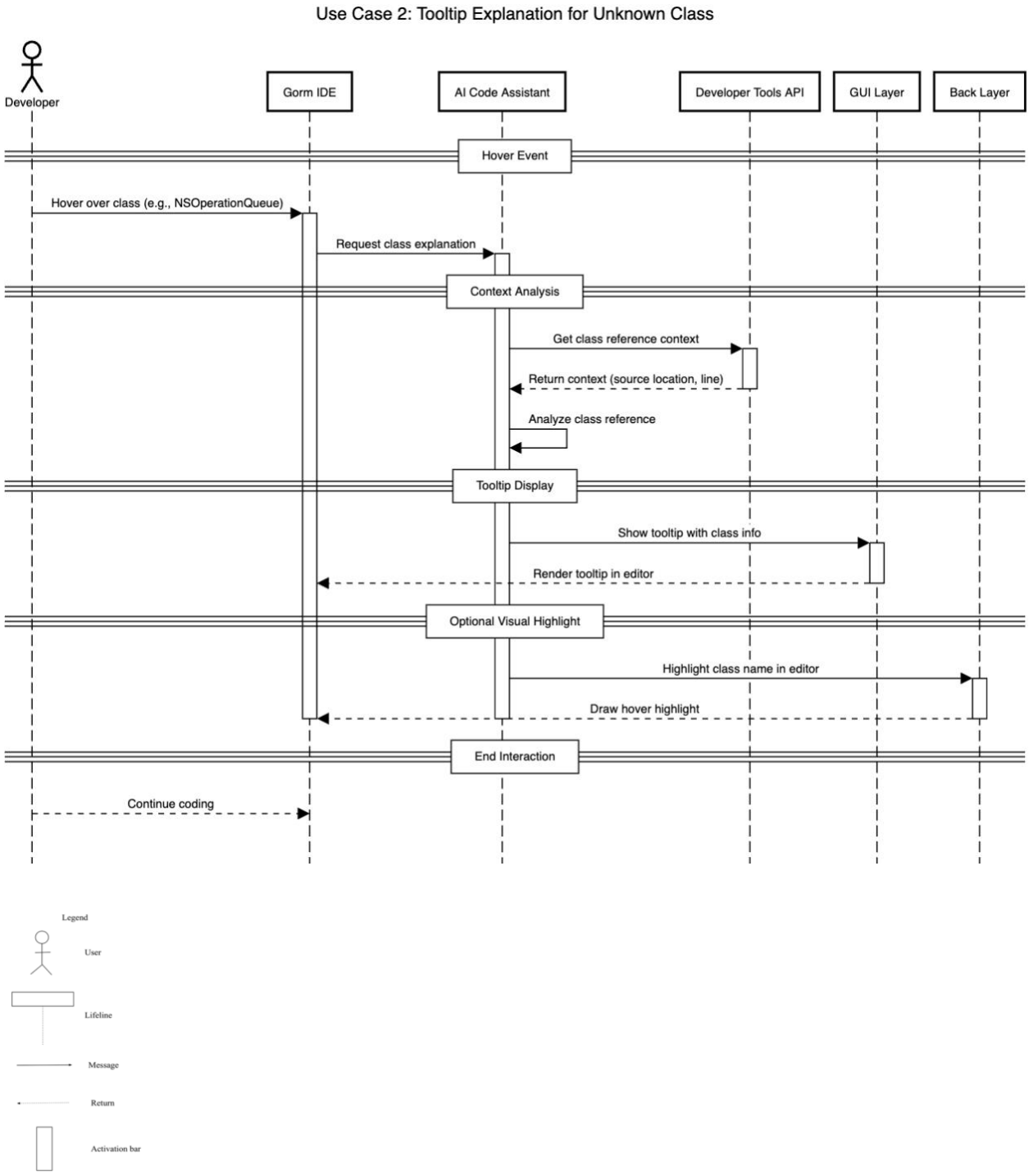



**Figure 4: UML Sequence Diagram of Use Case 2**

# 9 Plans for Testing the Impact of Interaction

To make sure the AI code assistant works well inside the GNUstep system, we designed several tests. These tests check if the assistant gives correct suggestions, does not break anything, and gives a good user experience. We divided our testing plan into different levels.

### 9.1 Unit Testing: GormController.m / GormController.h

We will test small parts of the assistant by themselves, to make sure they work correctly.

Prompt Builder:We will check if the assistant can build correct prompts based on selected UI elements. For example, when a user selects a button in Gorm, the prompt should include its name, type, and location.

Response Parser:After getting a suggestion from the assistant, this part needs to turn it into real Objective-C code. We will test different outputs to see if they are valid code, and check how the parser handles errors (for example, incomplete method, missing bracket).

### 9.2 Integration Testing

We will test how different parts of the system work together.

- Gorm ↔ Assistant:

We will test if selecting a UI element and clicking "Ask AI" sends the correct information, and if the suggestion is received and displayed correctly.

- ProjectCenter ↔ Code Insertion:

If users ask for a method suggestion in the code editor, we will test if it gets inserted in the right place in the file.

- Undo/Redo Support:

After a suggestion is added to the file, undo and redo should still work normally.

### 9.3 User Interaction Testing

We will test how different parts of the system work together.

- Gorm ↔ Assistant:

We will test if selecting a UI element and clicking "Ask AI" sends the correct information, and if the suggestion is received and displayed correctly.

- Suggestions Accuracy:

We will test if the assistant gives correct and useful suggestions in different situations (for example, button, label, table view).

- UI Feedback:

The suggestion should appear in a clear way. We will test if users can easily read the output, accept or cancel the suggestion.

- Preference Settings:

If the assistant is turned off in preferences, it should stay inactive. We will also test if changes to settings are saved properly.

### 9.4 Performance Testing

We want to make sure the assistant is fast and does not slow down the system.

- Response Time:

We will measure how long it takes from clicking “Ask AI” to getting a suggestion. If it takes too long, users might stop using it.

- Large Projects:

We will open a project with many UI elements and test if the assistant still works correctly and quickly.

### 9.5 Error Handling and Edge Cases

We will test what happens if something goes wrong.

- Invalid Input:

We will test how the assistant reacts when the selected object has no name or missing properties.

- Empty Suggestion:

If the assistant cannot give any output, Gorm should show a friendly message, not crash.

- Connection Problem:

If the assistant needs to reach an external service and fails, the system should show an error and continue working normally.

### 9.6 Security and Stability

Even if the feature is small, we still need to make sure it is safe and stable.

- No Data Leak:

User input should not be sent anywhere unless the user agrees in settings.

- System Crash Protection:

If the assistant crashes or fails, it should not crash Gorm or ProjectCenter. All errors should be caught safely.

## 10 Lesson Learned

While integrating the new AI Code Assistant into GNUstep, we gained a deep appreciation for the complexity of adding even a seemingly simple feature to a mature software system. We learned that such integration touches many parts of the architecture—from user-facing components like Gorm and ProjectCenter to backend elements like user preferences and configuration files. This experience highlighted the importance of understanding the role and interaction of impacted files to maintain system stability and organization. Initially, we considered embedding the assistant directly into existing tools for quick implementation, but we soon realized that a modular architecture—where the assistant functions as a separate subsystem connected through a clear API—offers greater long-term flexibility, reusability, and adherence to GNUstep's clean separation of concerns. Through this process, we also developed a more comprehensive view of software testing. Beyond unit testing, we considered integration, performance, and user experience, and we planned for edge cases like empty suggestions or connectivity issues. Ultimately, this experience helped us think more like real software architects and testers, emphasizing thoughtful design, modularity, and robust testing strategies when integrating emerging technologies into established systems.

## 11 Reference

GNUstep. (n.d.). Gorm documentation. GNUstep Wiki. Retrieved March 25, 2025, from

https://wiki.gnustep.org/index.php/Gorm

GNUstep. (n.d.). ProjectCenter overview. GNUstep Wiki. Retrieved March 25, 2025, from

https://wiki.gnustep.org/index.php/ProjectCenter

Stack Overflow. (n.d.). How to integrate code generation AI into IDEs? Retrieved March 25, 2025, from

https://stackoverflow.com/questions/72185789

Myers, G. J., Sandler, C., & Badgett, T. (2011). The art of software testing (3rd ed.). John Wiley & Sons.

Kim, J., & Nguyen, T. (2022). Design and evaluation of AI-powered code recommendation systems. Journal of Software Engineering Research and Development, 10(1), 1–15.

https://doi.org/10.1186/s40411-022-00123-5

GNUstep. (n.d.). ProjectCenter overview. GNUstep Wiki. Retrieved March 25, 2025, from

https://wiki.gnustep.org/index.php/ProjectCenter

GNUstep. (n.d.). Gorm documentation. GNUstep Wiki. Retrieved March 25, 2025, from

https://wiki.gnustep.org/index.php/Gorm